Towards Diversity of COTS Software Applications: Reducing Risks of Widespread Faults and Attacks

Marco Casassa Mont, Adrian Baldwin, Yolanta Beres, Keith Harrison,
Martin Sadler, Simon Shiu
Trusted E-Services Laboratory
HP Laboratories Bristol
HPL-2002-178
June 26th , 2002*

E-mail: marco_casassa-mont@hp.com, adrian_baldwin@hp.com, yolanta_beres@hp.com, keith_harrison@hp.com, martin_sadler@hp.com, simon_shiu@hp.com

COTS applications, diversity, faults, attacks, survivability, security, trust

Recent IT attacks demonstrated how vulnerable consumers and enterprises are when adopting commercial and widely deployed operating systems, software applications and solutions.

Diversity in software applications is fundamental to increase chances of survivability to faults and attacks.

Current approaches to diversity are mainly based on the development of multiple versions of the same software, their parallel execution and the usage of voting mechanisms. Because of the high cost, they are used mainly for very critical and special cases.

We introduce and discuss an alternative method to ensure diversity for common, widespread software applications without requiring additional computational resources. This method takes advantage of the componentisation of modern software solutions and enforces diversity at the installation time, by a random selection and deployment of critical software components. Randomisation criteria are adaptable to feedback gathered from software installations and affect software components' lifecycle. We describe a few encouraging results obtained from simulations.

* Internal Accession Date Only
Approved for External Publication

1. Introduction

In the last two decades, commercial software has gone through a process of consolidation and homogenisation. The current commercial computing environment, both within the enterprise and home, is largely dominated by a few software systems, at the operating system (OS) level (e.g. Microsoft Windows, Linux, Unix, etc.), software development level (software frameworks such as Java and Microsoft .NET), application level (application suites such as Microsoft Office, etc.) and Internet access level (such as browser and web servers like those provided by Netscape, Microsoft, etc.)

On one hand this process has lowered the costs of products because of the economy of scale and provided common platforms to simplify interactions. On the other hand there has been an increasing number of widespread attacks exploiting vulnerabilities of massively deployed software. Recently, code red [22], code blue and Nimda [23] worms caused huge problems to corporations and individuals by exploiting simple software vulnerabilities like the buffer overrun bug [8]. Large populations of users, employees and business have been affected causing economical and social problems.

Software bugs and vulnerabilities have such a dramatic impact because the large number of identical installations makes it easy to exploit these faults as attacks and hence the absence of diversity increases the exposure of most systems on the Internet.

Unfortunately software bugs are inevitable in most, if not all, software systems, especially with the current levels of complexity. The adverse effects of such bugs vary in severity but all are generally capable of causing faults and malfunctions and some can leave the software system vulnerable to external attacks. In view of the fact that every user's installation of a specific software is identical, each installation will include the same bugs, and therefore vulnerabilities. As a result, large scale attacks on software systems are successful because computer hackers are likely to make the (correct) assumption that most, if not all, of the targeted operating systems or software applications are built in exactly the same way and, as such, have the same bugs and problems. Attacks can be tailored to each system, but recent viruses such as code red have caused untargeted systems, such as Internet enabled printers, to crash even though they are not the intended target. Similarly, a major fault or malfunction caused by a bug in the software system will affect all users in the same way.

Concern has been expressed in the agricultural industries as the genetic diversity of crops is reduced to allow particular pesticides to be used. This can have the effect of reducing the resistance to particular diseases or where a disease strikes it can wipe out an entire crop. Analogies can be drawn to the eco-system of computers on the Internet where viruses evolve much quicker than systems change, yet the large number of identical systems enables a virulent virus to spread very quickly thus causing significant damage. Therefore it is believed that diversity is fundamental to prevent faults and attacks.

Critical and special-purpose software and applications (like the software systems controlling nuclear power stations, aircraft and spacecraft, bank exchanges, etc.) are

designed, implemented and deployed by keeping in mind the importance of ensuring operational survivability and reliability. These requirements are usually met by adopting very expensive solutions based on replication and independent software and systems.

Unfortunately, the approaches used for critical software are not suitable for common and widespread operating systems, software and applications mainly because of the involved costs, the implications in term of economy of scale, the need for additional computational resources and the peculiarity of the targeted market.

Despite this, we believe that diversity can also be achieved for common and popular software applications in respect of their cost effectiveness and constraints on required computational resources.

In this paper we briefly describe some current techniques and mechanisms used to ensure diversity in software applications. We then introduce and discuss an alternative approach to software diversity aiming at the reduction of widespread software attacks and faults. This approach takes advantage of the componentisation of modern software solutions and enforces diversity at the installation time by randomly selecting and deploying critical software components.

2. Software Diversity: Background and Requirements

The problem of dealing with faults and attacks for information, software and systems has been widely analysed and researched in the past.

Software diversity is a key element to achieve protection [1] against both natural phenomena (including random failures, physical damages and corrupted information) and human actions (including design faults, interaction faults, malicious logic, intrusions and physical attacks).

2.1 Related Work

N-version software diversity has been analysed and proposed [2], [18], [19] as a means of dealing with uncertainties of design faults. The basic concept is that having N independently developed versions of the software minimises the likelihood of coincident failures and vulnerabilities. The system is then built from these (three or more) separate software versions with a decision algorithm, for example a majority vote, determining the overall result.

Diversity can be enforced not only at the software design level but also at the functional level [9]. Functional diversity is a way of forcing multiple design teams to be "intellectually diverse" in their solutions to the design problem.

The N-version technique has mainly been adopted for critical and special-purpose cases, like software for flight control computers [3], [4], and design of nuclear reactor protection systems [5], [15] because of the high costs involved.

The main objective of most of the work done on diversity is to achieve a higher reliability of software applications [16]. Whether diversity is a convenient means for delivering high reliability has been subject of debates and discussions [17].

Recently, diversity has also been investigated from the perspective of populations and ecosystems of software systems. Relevant research has been done on survivable systems, i.e. systems characterised by the ability to provide essential services even in the presence of intrusions and faults and recover full services in a timely manner [6]. Specifically, [7] describes systematic techniques to improve resistance to intrusions and attacks by diversification of system software, thereby increasing the cost and difficulty of identifying vulnerabilities. The approach is based on stochastic diversification and it is achieved by transforming a program into several versions each with additional logical complexities that obscures the behaviour whilst maintaining correct function and performance.

2.2 Requirements

The core problem addressed by this paper is enforcing diversity for widespread commercial of the shelf software (COTS) in order to reduce the risks of large-scale attacks and other failures.

We target large and homogeneous populations of commercial software installations, commonly used for day-by-day business and consumer tasks. Examples of these populations include enterprises (large number of employees' PCs having the same software install-base), Internet communities of people and organisations sharing similar interests.

In this context, the problem of making a specific software installation survivable to a fault or an attack is secondary to the problem of minimizing the effects as an attack spreads and maximizing the number of working systems within the population.

The impact of an attack or a fault on commercial software on a single installation is usually minimal especially when common security policies (like periodic data backup, virus checking, etc.) are put in practice. On the contrary, it is the transmission of attacks over a larger population, in a short period of time, that creates the serious economical and social damage; for example, it can cause the interruption of network and e-mail communication, leading to the interruption of business processes. A further issue is the clean up costs where considerable effort from technicians is required to stop viruses and worms from spreading by applying patches and recovering from compromises.

The basic requirements for diversity in common commercial software can be summarised as:

- Provide mechanisms to avoid faults and attacks that quickly propagate over a large population of installations;
- Preserve the relatively low costs of COTS (due to the economy of scale);

- Avoid the need for extra computational resources.

Special-purpose solutions traditionally used in the N-version approach do not fulfil those requirement and they represent an over-engineered approach to the specific problems addressed in this paper. It is also not really clear if commercial software developers are willing to embrace diversity techniques based on obfuscation of the deployed code [7].

Next section describes an alternative approach based on existing mechanisms for the design and development of software systems. This approach introduces an element of diversity at the deployment time, without requiring any modification of the deployed code or additional computational resources.

3. Proposed Approach

The approach proposed in this paper exploits the componentisation and object-oriented aspects of modern software: current operating systems, software applications and solutions are built from software components, each of them implementing specific well defined functionality.

Software engineering techniques dealing with software life-cycle management have been around for years and are commonly used during software development projects. For example, tools for software modelling, based on UML [10] or similar techniques, provide mechanism to model, design, refine, implement, test, deploy and maintain complex software systems and applications.

Specifically complex software applications can be analysed from structural and behavioural aspects, different views can be provided at different levels of abstraction, ranging from high level classes and objects (and their relationships) to the physical software components that are going to be deployed.

During software design and development, designers and engineers should also go through risk management activities, which include: identify critical software components, their vulnerabilities to potential attacks and faults, and mitigate the involved risks. The methodology for identifying critical components would be different from traditional critical system tasks. It may be that the complex algorithms at the heart of the system are considered critical and therefore must be well engineered. It should also be recognized that the most vulnerable components are also highly critical – this suggests that external-facing components should be considered critical. It is software bugs in these external-facing components that often become subject to attacks such as buffer overflow attacks providing viruses and hackers with a way into the system.

The critical components are not necessarily those directly developed as part of an application. As application development frameworks become more advanced and include many more base libraries (such as Java and Microsoft's .Net) bugs in these underlying libraries could negate the advantages of diversity. Diversity could be introduced at the level of these frameworks as well as, or instead of, at the application layer.

The proposed model makes use of multiple implementations of critical components. Because of the separation of concerns between the design and the implementation phases, modern software development tools allow the development of multiple implementations of the same software component, in a way that is compliant with defined interfaces. We relax the constraint of having multiple implementations of the whole software applications (as mandated by most of the N-version techniques) as we concentrate the effort only on critical components.

3.1 Model

A commercial software application is generally supplied by a software provider as a package on some form of storage medium, including its components and installation software which, when run on the customer's computing environment, installs the various components for future use. The individual components included in each package are generally identical to components on other packages provided to other customers. Usually the result is that all the software installations are substantially identical. The user may install different options and various patches and service packs bringing a degree of diversity; however, many corporate systems will have a software repository where the company standard is issued with standard options and patches.

In our model, we introduce an element of diversity at the installation time by modifying the installation process. Multiple implementations of critical components are available in the installation package. For each critical component a software installer randomly selects and installs one of the available implementations. Figure 1 shows the model of a system implementing this approach:

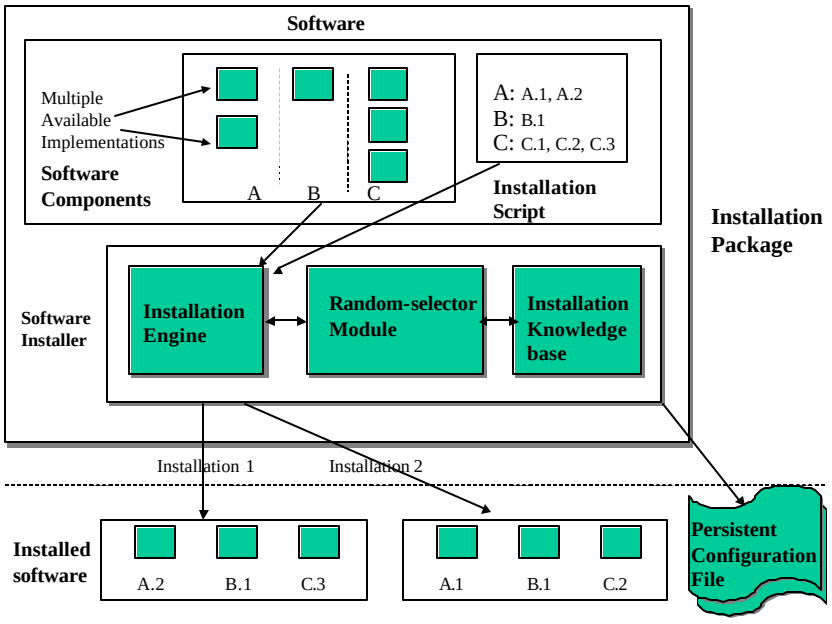

Figure 1: Model

Software is distributed by means of an installation package which include three basic parts:

- A software components bag;
- An installation script;
- A software installer.

The *software components bag* contains the components used to form the software application. Software components might include COM components, EJB components, .dll libraries, .exe executable files, configuration files, etc. For each critical component multiple implementations are available. For example, in Figure 1, components A and C are critical. Two implementations are available for component A and three implementations are available for component C.

The *installation script* contains the necessary information to successfully install the software application, including the list of all the available components, the installation sequence and dependency constraints.

The *software installer* is the core part of the installation package. It contains three modules:

- Installation Engine;
- Random-selector module;
- Installation knowledge base.

The *installation engine* is in charge of interpreting the installation script and installing the software application. This engine interacts with a *random-selector module* each time a critical component (having multiple implementations) has to be installed.

The *random-selector module* is driven by a random-function that, given a critical software component, randomly selects one of the component implementations at random. This function can be constrained by information contained in the installation knowledge base.

The *installation knowledge base* is a local database containing contextual installation information. This information might include the status of other installations and the evolution of a particular installation over its lifetime (including changes due to patches, upgrades or maintenance). It may also include known bad combinations where components have known faults when installed on particular OS versions.

In particular contexts, like enterprises and large organizations, a variant of our model can be used to install a particular software system on a number of computers. In this situation, the selection of critical software components to be installed may depend upon which implementations of components have previously been installed on other computers. The information necessary for making such decisions is stored in the installation knowledge

base. This can ensure that there is sufficient diversity in a computing environment; for example, a server farm thereby ensuring a degree of resilience.

Each installed software application has a proper identity defined by the sequence of the installed components. This sequence is a sort of e-DNA. The installer stores this sequence in a local *persistent configuration file* along with a copy of the installation knowledge base.

Another variant of the model uses an installation mechanism provided by a centralized installation service, for example within an enterprise. This approach facilitates the collection and management of configuration information associated to each installation for future software maintenance or upgrades.

After the installation process, for security reasons, the software installer makes sure that implementations of critical components that have not been installed are deleted from the platform where the software is installed.

3.2 Properties

The proposed model introduces an element of diversity into the software at installation time without the constraints of the traditional N-version software. It is not as expensive or impractical as the N-Version approach as it does not require several distinct full implementations of the same software and their parallel executions. However, it protects a population of systems rather than any particular system and as such does not provide a solution for safety critical systems.

Not all the components need to have multiple implementations. At the end of the installation phase, a copy of the software application is installed as usual but with a potential unique combination of software components. Every installation of the same software application is potentially different but its functionalities, interfaces and expected behaviour are the same. The degree of diversity directly depends on the number of critical components, the number of available implementations and the selection criteria in the random function.

This approach does not prevent a specific installation of an operating system or software application from being subject to fault or being attacked: it is likely that components will still have software bugs and vulnerabilities. Nevertheless, it reduces the risk of massive propagation of faults and attacks to large population thanks to the intrinsic diversity of each installation. With this approach it is also less likely that two or more installations of the same software will crash due to the same fault, at the same time, when executing similar operations.

Hacking techniques taking advantage of bugs in specific component (or due to the combination of specific components) may gain information from a specific installation but the chances of this being applicable to other systems (using different components) is very much reduced.

4. Experiments

The discussion so far has claimed that adding diversity into a population of systems increases its robustness, particularly when attacked by viruses that take advantage of common bugs. A simulation of the spread of a virus has been carried out to demonstrate some of the properties that increased diversity would achieve.

The simulator created a number of virtual machines, each with its own IP address, and a list of components, along with implementations (versions) of each component. A virus with a propagation mechanism similar to code red [22] was then simulated where the virus infects by using a bug in a particular version of a component. Once a machine is infected the virus tries to spread to other machines by generating IP addresses at random according to the current machines sub-mask; thus the probability of picking local machines is high but there is a sufficient chance of IP addresses outside of the local network to ensure a world wide spread. The virus then pings the other IP addresses and attempts to infect those it finds using the same bug. Each infection tries to infect 200 other machines and then remains dormant – in the case of code red a security hole allowing access to all files remained in place.

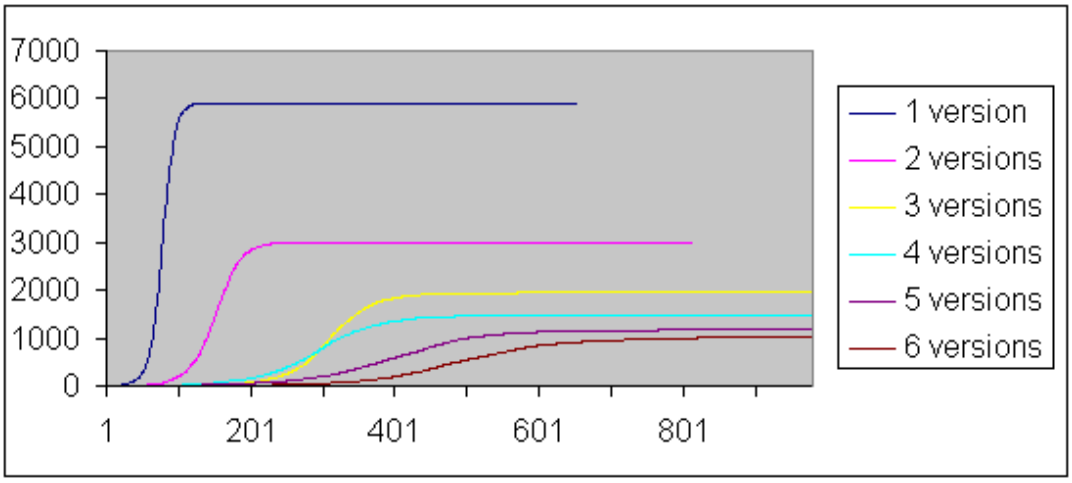

Figure 2: Experiment 1 – Increasing the diversity of components where a virus attacks a single version

The first experiment simulated 6000 systems on a sub-net. A number of simulations were run with increasing diversity, from 1 to 6 versions, in the component targeted by the virus. Figure 2 shows the variation in the rate of infection over time. Two factors are worth noting: firstly since only one version of the component is being infected the final number of infections is inversely proportional to the number of components' implementations; secondly the rate of infection is slowed as it becomes harder to find susceptible systems. It is also worth noting that some diversity can be very valuable but as the diversity increases the rate of slowdown in infection rates decreases and as such there are clearly diminishing returns.

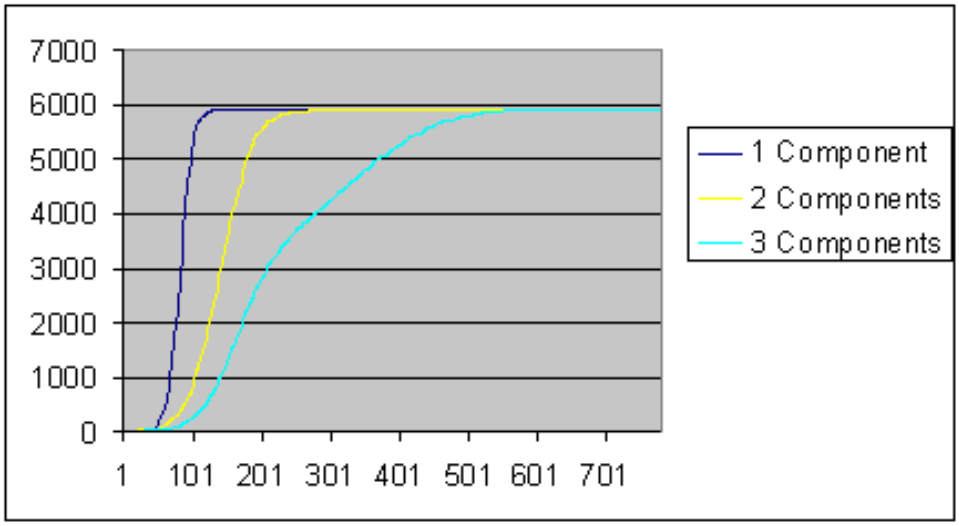

Figure 3: Experiment 2 - The effects of diversity when all components are vulnerable to separate viruses.

The second experiment was carried out on the same set of systems but looked at the effect of having viruses attacking all versions of the components. Separate viruses were created to attack a component with 1, 2 and 3 implementations. Figure 3 shows the infection rates over time. The infection rates do saturate although increasing the number of components implementations does delay the rate of infection and it also delays the peak in network traffic due to the virus by a corresponding amount. This delay in infection rate is due to the reduction in the probability of finding a vulnerable system and hence will be inversely proportional to the number of implementations of a component. This gives system administrators a larger window in which to clean up machines and install the necessary bug fixes.

These experiments show that there is a clear advantage to increasing diversity of standard components to help in managing attacks. It is clear from the results that both the number of infected systems and the speed of infection are inversely proportional to the level of diversity. It is worth noting that a composite virus such a Nimda [23] that infects via many software bugs will increase the infection rate. It is clear that a small amount of diversity in many standard components will bring considerable gains but after that the returns will diminish.

5. Discussion

The feasibility of the proposed model has to be validated against real-world scenarios. Section 6 describes our plans for tests and further experiments while this section discusses general software engineering and operational aspects relevant to the model.

The proposed model does require the development of multiple versions of software but it restricts this requirement to critical components. Even if it relaxes the constraints introduced by the classic N-version approach, particular attention has still to be paid during two critical phases:

- Risk analysis for potential vulnerability and subsequent identification of critical components;
- Software testing phase.

If the risk analysis phase is not properly executed, the misjudgement of which components are critical could seriously compromise the effectiveness of the diversity introduced at installation time. On the other hand, an extended usage of this technique (by including components that potentially are not critical) might increase the overall complexity of writing and maintaining the software and the associated costs.

The software-testing phase must include white and black box testing activities for each implementation of a software component. Modern software engineering and development tools provide mechanisms to define interfaces and behavioural specifications for software components. Multiple implementations of each software component should be tested against those specifications.

Testing all the possible combinations of the software components can be extremely expensive. On one hand the fact of having a large set of possible combinations of software components is the strength of this approach. On the other hand it introduces complexity. The testing phase of the complete software application can still be done on an empirical base, by testing a reasonable set of installations of the software, generated in a random way. By doing so, particular faulty combinations of software component implementations can be detected in advance and avoided during the installation of the software (by storing this information in the installation knowledge base of the installation package).

Gathering knowledge from software installations is extremely important for software producers, not only during the testing phase but also during the whole software lifecycle (maintenance, upgrades, etc.). It is important for a software producer to collect information about bugs and undesired behaviours from the population of software installations in order to correct faults and avoid the occurrence of faulty combinations of components in future installations. This task is simplified by the fact that each software installation has an identity (its e-DNA) describing the particular combination of deployed components.

The information collected by monitoring for problems and issues related to deployed components can ultimately be used to make decisions about the destiny and evolution of specific components (modify, extend, abandon, etc.) or combinations of components. In large enterprises and organisations the task of monitoring large population of software installations can be delegated to traditional IT support centres, who can then interact with software providers.

Definitely, the software installer module plays a key role in ensuring a correct installation of software components and the enforcement of particular installation policies. It is a trusted module. The overall installation package must be properly secured to guarantee its integrity and trustworthiness (by digitally signing its code and potentially obfuscating its modules). If centralised within an enterprise or organization, the software installation service plays the role of a trust service [11] and it must be accountable during software installation, information gathering and maintenance management.

The proposed approach to software diversity is potentially suitable not only for traditional software producers but also for open source software. In both cases it is important that component interfaces and expected behaviours are clearly defined and specified at design time. Specifically, the open-source initiative can take advantage of the willingness of lots of participants to contribute to the development of software solutions: multiple implementations of software components can be made available in software packages and installed using our approach.

6. Current and Future Work

In addition to the experiments made by simulations, we are also investigating the feasibility and effectiveness of our model by means of practical experiments involving widely distributed software applications. Our tests will include experiments with software applications that provide long-term storage of digital documents [20] and distributed software agents that support storage and replication of data [21].

We are planning to re-develop these applications by providing at least two different implementations for each critical component and create multiple populations by deploying such applications, including one where applications are deployed in a classic way, without diversity. Experiments are going to help us to better understand the effects of the random aggregation of components at the deployment time, measure the efficacy of the random selection module and understand the feasibility of adaptation mechanisms. We are also going to observe and measure for real the effects of attacks (exploiting vulnerabilities introduced by software bugs) on populations created by using our diversity approach and compare them against a population deployed in a conventional way.

In terms of future work, we are planning to investigate the feasibility of our approach for advanced e-commerce scenarios, whereby multiple implementation of core e-services (like electronic payment services, billing services, booking services, etc.) are available to consumers and enterprises (for example by using UDDI servers [12]) and are composed on-the-fly [13], [14] to obtain added-value e-services. In such a context the composition of web services will happen by randomly selecting and aggregating core web services with equivalent functionalities and compliant with user's requirements (contractual clauses, specifications, QoS policies, etc.)

7. Conclusion

Today it is of primary importance to deal with lack of diversity in widely deployed commercial software as faults and attacks quickly spread across large population of identical installations creating enormous economical costs.

Current approaches to diversity, based on multiple versions of the same software, potentially running in parallel on different computational resources, are too expensive and are mainly used in critical and special-purpose cases.

This paper introduces an alternative approach to diversity which takes advantage of the componentisation of modern commercial software. Critical software components are identified during the risk assessment phase and multiple (functionally equivalent) implementations are developed. These multiple implementations of components are distributed within software installation packages. At the installation time, an installation module randomly selects and installs an implementation of each critical component, in respect of potential pre-defined constraints and policies.

The proposed system can take account of problems encountered in a large population of installations of the same software application. Components might evolve during their lifetime. Criteria for randomly selecting software components can adapt dynamically so that problematic combinations of components are avoided and specific faulty components are modified or banned.

Software developers need to clearly specify software component interfaces, their behaviour and identify critical components by assessing their vulnerabilities and the involved risks.

Our experiments based on simulations show that there is a clear advantage to increasing diversity of standard components to help in managing attacks. It is clear from the results that both the number of infected systems and the speed of infection are inversely proportional to the level of diversity. The feasibility and efficacy of the proposed model has to be verified in real-world scenarios.

8. References

[1] A. Avizienis – Design Diversity and the Immune System paradigm: Cornerstones for Information System Survivability – ISW2000 – 2000

[2] A. Avizienis, L. Chen – On the implementation of N-version Programming for Software fault Tolerance during Execution – COMPSAC 1977, Chicago – Nov 1977

[3] Y.C. Yeh - Dependability of the 777 primary flight control system. – Dependable Computing for Critical Applications 5, pages 3-17 – IEEE Computer Society Press - 1998

[4] D. Briere, P. Traverse – Airbus A320/A330/A340 electrical flight controls: a family of fault-tolerant systems – Digest of FTCS-23, pages 616-623, - June 1993

[5] P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti - PODS - A Project on Diverse Software – IEEE Trans. Software Engineering, Vol SE-12, No. 9 - 1986

[6] R.J. Ellison, D. Fisher, R.C. Linger, H.F. Lipson, T. Longstaff, N.R. Mead – Survivable network Systems: An Emerging Discipline – Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU/SEI-97-032 - 1997

[7] R.C.Linger - Systematic Generation of Stochastic Diversity as an Intrusion Barrier in Survivable Systems Software - Carnegie Mellon University, Pittsburgh, PA – 2001

[8] A.K. Ghosh, T. O'Connor – Analyzing Programs for Vulnerability to Buffer Overrun Attacks - Proceedings of the 21st National Information Systems Security Conference, Crystal City, VA - October 5-8, 1998

[9] B. Littlewood, P. Popov, L. Strigini, - Design Diversity: an Update from Research on Reliability Modelling, Proc. Safety-Critical Systems Symposium Bristol, UK – 2001

[10] G. Booch, J. Rumbaugh, I. Joacobson – The Unified Modelling Language User Guide – Addison Wesley - 1999

[11] A.Baldwin, Y.Beres, M. Casassa Mont, S. Shiu – Trust Services: Reducing Risk in E-Commerce, ICECR-4 - 2001

[12] B. McKee, D. Ehnebuske, D. Rogers – UDDI Version 2.0 Specification - June 2001

[13] F. Casati, S. Ilnicki, L.J. Jin, and M.C. Shan - An Open, Flexible, and Configurable System for Service Composition - In Proceedings of the Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000), pages 125-132, Milpitas, California - 2000

[14] G. Piccinelli, L. Mokrushin - Dynamic e-service composition in DySCo - IEEE DDMA – 2001

[15] A.E. Condor, G.J Hinton – Fault tolerant and fail-safe design of CANDU computerised shutdown systems – IAEA Specialist Meeting on Microprocessors important to the Safety of Nuclear Power Pants, London – 1988

[16] G. Dahll, J. Lahti – An investigation of methods for production and verification of highly reliable software – Proceeding SAFECOMP'79 – 1979

[17] L. Hatton – N-Version Design Versus one Good Version – IEEE Software, 14, pages 71-76 – 1997

[18] J.C. Knight, N.G. Leverson, L.D.S Jean – A Large Scale Experiment in N-Version Programming – Proceedings 15th International Symposium on Fault Tolerant Computing (FTCS-15), pages 135-139 – 1985

[19] M.R. Lyu, Y. He -Improving the N-Version Programming Process Through the Evolution of a Design Paradigm - IEEE Trans. on Reliability, Sp. Issue on Fault tolerant Software, vol. 42, no. 2, pages179-189 - 1993

[20] A. Baldwin, M. Casassa Mont, S. Shiu, A. Norman - PAST Service Permanent Active Storage Service: Survivability - HPL-2001-203 – [HP Restricted] - 2001

[21] M. Casassa Mont, L. Tomasi - A Distributed System, Adaptive to Trust Assessment, based on Peer-to-Peer Evidence Replication and Storage - FTDCS'01 – 2001

[22] CAIDA – CAIDA Analysis of Code Red, Code-Red Worms: A Global Threat - http://www.caida.org/analysis/security/code-red/ - 2001

[23] SANS Institute – Nimda Worm/Virus Report, Final – source incidents.org - http://www.incidents.org/react/nimda.pdf - October 2001